Using a Subsumption-Based Taxonomy to Construct Networks of Cooperating Decision Procedures

Pierre Lim, Mark Wallace
European Computer-Industry Research Centre
Arabellastraße 17
8000 Munich 81
Germany
{lim,mark}@ecrc.de

August 1992

Abstract

Costly general decision procedures can be sped up by incorporating efficient decision procedures for restricted classes of constraints. We show how to make use of these specialized decision procedures by constructing a network of cooperating decision procedures. Of course, in order to gain efficiency the expensive *general* decision procedures should not be called as far as possible. But they *should* always agree with the answers produced by the specialized decision procedures. Another problem is that each specialized decision procedure can handle only a certain class of constraints and hence the decision made by any *one* specialized decision procedure is based on a subset of all the constraints actually there. Nevertheless this *local* decision making *can* be used in short cutting the decision making process. This is accomplished by defining (i) a network topology, and (ii) a constraint distribution scheme, such that there is guaranteed to be *consensus* amongst the decision procedures. In this way when a specialized decision procedure decides *local* consistency (or inconsistency) then indeed we have *global* consistency (or inconsistency).

Topics: algorithms, reasoning architectures, constraint solving, classification

1 Introduction

In this paper we study the problem of how to efficiently decide the consistency of a set of constraints. Our objective is to speed up *general* decision procedures by using *specialized* decision procedures to decide restricted classes of constraints efficiently. This is achieved by arranging the decision procedures into a network such that consensus is *always* achieved. Note however that this is *not* an exercise in distributed problem solving but rather what our network specifies is a means for *reducing* cost by *not involving* expensive decision procedures as far as possible.

Our work is carried out in the context of Constraint Logic Programming (CLP) [14]. CLP is a natural implementation vehicle for hybrid knowledge representation languages. Terminological concepts are often expressed as constraints [7]. The implementation of Krypton extended the basic inference rule of the assertional reasoning mechanism by "altering the meaning of unification" [1]. With logic programming supporting the assertional component of a hybrid system, and the constraints the terminological component (as in [19]), CLP offers precisely the right architecture since it provides logic programming with unification replaced by constraint solving. Given an underlying computation domain D, we assume a CLP(D) implementation involving a set of (specialized) decision procedures. It is *not* the purpose of this paper to show how to build *the* general procedure by combining the specialized ones (this topic is discussed by [18, 17]), rather we *assume* the existence of the general procedure. The problem *we* consider is how to use the existing decision procedures as efficiently as possible.

CLP programs comprise (i) programmer-defined predicates which are evaluated using SLD-resolution, and (ii) constraints (atoms where the principal functor has a fixed interpretation) which are decided by some builtin decision procedures. Consider the following program for performing complex arithmetic [15].

```
zmul(c(R1,I1), c(R2,I2), c(R3,I3)) :-
    R3 = R1 * R2 - I1 * I2,
    I3 = R1 * I2 + R2 * I1.
```

The goal `zmul(c(1,1),c(2,2),c(X,Y))` produces the answer `X = 0, Y = 4`. With a certain combination of variables instantiated, e.g. `R1` and `I1`, the two arithmetic equalities are linear whereas with certain other instantiation patterns, e.g. `R3` and `I3` the constraints are nonlinear. If the constraints are linear then it is definitely much more efficient to use a linear programming method. In terms of theoretical complexity Khachiyan [16] has shown the linear programmming problem to be polynomial whereas the problem of deciding arbitrary real arithmetic constraints has been shown to be doubly exponential [5]. But if the set of constraints contains both linear and nonlinear constraints then during computation if a new *linear* constraint is to be checked for consistency with the collected constraint set then it would be advantageous to be able to determine if this check can be done *only* with the linear solver. That is, the consistency check is done by *only* checking the *linear* constraints. Of course, we have to be careful since in general there will be interactions between the linear and nonlinear constraints and such local consistency checks will not imply global consistency.

Our aim is to develop a framework for building networks of decision procedures such that a check of a *local* constraint store is sufficient to decide global consistency. The problem we address in this paper is to arrange the partial information we have (represented by constraints) in such a way that interactions between decision procedures and constraints are made explicit. Once this information is explicitly available it is then possible to perform a number of optimizations. We give several examples of these. Additionally, we use our framework to analyze a complex decision mechanism (the multi-layered decision procedure of CLP($\mathcal{R}$)).

2 Formalization

The CLP Scheme of Jaffar and Lassez generalizes logic programming by replacing the notion of unification with that of constraint solving over a specified domain [14]. A key idea here is that certain relation and function symbols have a *fixed* interpretation given by a structure of computation. This means that specialized domain-specific decision procedures can be used, e.g. Gaussian elimination for deciding real linear equations and the Simplex method for deciding real linear inequalities. However, there are many other methods such as forward checking [8] which can decide more restricted systems of constraints more efficiently. It is desirable to incorporate these algorithms into more general ones to take advantage of special-case solving to obtain speedups.

The fixed interpretation in CLP languages gives rise to an algebraic semantics which is equivalent to the model-theoretic, proof-theoretic, fixpoint and operational semantics. We shall use the proof-theoretic characterization to explain our scheme since this greatly simplifies the exposition.

2.1 Specialized Decision Procedures

As required within the CLP Scheme, a decision procedure for a class of constraints takes as input a set of constraints of that class and decides if they are consistent.

In this paper we shall be concerned with networks of *specialized* decision procedures. Each specialized decision procedure D_i has an associated *local store* S_i which is a set of constraints of the appropriate class, $\mathcal{C}(D_i)$. Some specialized decision procedures can also decide if a new constraint is implied by a given set of constraints. A decision made by D_i is either that a new constraint $C \in \mathcal{C}(D_i)$ is inconsistent with S_i or that it is implied by S_i. For each computation domain D, the implementation of a $CLP(D)$ system requires a network of (one or) more specialised decision procedures which are used in such a way that they support an (abstract) decision procedure for D.

This paper addresses the problem of efficiency. We seek to obtain a "globally correct" decision, by using the cheapest specialized decision procedures possible. We seek to place new constraints in the local stores in the best way so as to ensure that local decisions are, as often as possible, globally significant. We seek to establish communication between the specialized decision procedures which minimises the cost of current and future decisions.

To this purpose we introduce the notion of an "answer" returned from a specialized decision procedure. Our fundamental requirement is that the answers returned from all the decision procedures in a network should agree. This property we term "consensus".

2.2 Definitions

In this section we introduce some definitions and terminology which are in general use throughout the paper. More specific definitions will be presented in the context where they arise.

Terms are built in the usual recursive manner respecting signatures. A term in which the principal functor has a fixed interpretation is called an *atomic constraint*, e.g. in the domain of

uninterpreted functors over real arithmetic terms the constraint relation symbols are $\{=_{Herbrand}, =_{real}, \leq, <, >, \geq\}$. The programmer may *define* relations using *uninterpreted relation symbols* such as `gcd` or `append`. Well-formed formulae are formed from terms using connectives and quantifiers in the usual manner. We call a conjunction of atomic constraints a *constraint*. The *constraint programming languages* we shall consider use just the Horn clause subset of predicate logic and conform to the well-known syntactic conventions of PROLOG.

2.3 Logical Semantics

Although the constraints we are interested in are often described in terms of algebraic structures we are more interested in their *behaviour*. For us then, a better characterization is in terms of theories. The link between the structure and the theory is given by Jaffar and Lassez [14] for the logical semantics of CLP(D). They introduce a theory T_D (a set of formulae closed under logical consequence) which plays the same role as the computation domain (structure) D. T_D is required to satisfy two conditions:

- $D \models T_D$
- for all finite constraint stores $S \subset \mathcal{C}(D)$, $D \models S$ implies $T_D \models S$.

To extend their results to general logic programs, with negation, Jaffar and Lassez imposed an extra condition on T_D called satisfaction completeness [14]. This condition holds for a theory T with respect to a class $\mathcal{C}$ of constraints, if for any constraint $C \in \mathcal{C}$, whenever C is not inconsistent with the theory, $\exists C$ is a logical consequence of it. In other words, whenever it is not the case that $T \models \forall \neg C$, then $T \models \exists C$.

Let us take as an example the basic theory T_{eq} for an equivalence relation eq:
$\forall X.eq(X, X)$
$\forall X, Y.(eq(X, Y) \rightarrow eq(Y, X))$
$\forall X, Y, Z.(eq(X, Y) \wedge eq(Y, Z) \rightarrow eq(X, Z))$
This theory is satisfaction complete, for a rather trivial class of constraints which admits unquantified formulae with predicate eq and no constant or function symbols. However if we extend the class of constraints to admit function symbols, it is no longer satisfaction complete, because there are constraints on eq, such as $eq(g(X), f(X))$, for which neither $T_{eq} \models \forall X.\neg eq(g(X), f(X))$ nor $T_{eq} \models \exists X.eq(g(X), f(X))$.[1]

2.4 Formalising Decision Procedures

Under the logical semantics for $CLP(D)$, a decision procedure should decide whether for a finite set of constraints $S \subset \mathcal{C}(D)$:
$T_D \models \forall.\neg S$
(where $\neg S$ denotes the negation of the conjunction of the constraints in S), or if

[1] However it can be extended in turn to a satisfaction complete theory T_{Ceq} for unquantified eq constraints with function symbols by adding Clark's equality axioms [3]. The extended theory yields $T_{Ceq} \models \forall X.\neg eq(g(X), f(X))$.

$T_D \models \exists.S$

Operationally in a $CLP(D)$ system, the constraints are added incrementally. In a given computation state the current constraint store S is always consistent, and the decision procedure is required to decide if for a new constraint C, $T_D \models \forall.(S \rightarrow \neg C)$. Whenever this is not the case, it follows by satisfaction completeness that $T_D \models \exists.(S \wedge C)$

We give a similar logical semantics for each specialized decision procedure D_i by associating with it a logical theory T_{D_i}. If S_i is the local constraint store associated with D_i, and $C \in \mathcal{C}(D_i)$ is a new constraint, then local inconsistency is formalised as
$T_{D_i} \models \forall.(S \rightarrow \neg C)$
The new constraint C is locally consistent if it is not locally inconsistent. Notice, however, that the specialized theory T_{D_i} is not necessarily satisfaction complete, so it does *not* follow that
$T_{D_i} \models \exists.(S \wedge C)$.

The network of specialized decision procedures implementing a $CLP(D)$ system must combine to yield a global (abstract) decision procedure for D. In this paper we assume the specialized decision procedures are both correct and complete with respect to the global decision procedure. Formally correctness follows just if $T_{D_i} \subseteq T_D$ for each specialized decision procedure D_i. Completeness follows if for any atomic $CLP(D)$ constraint C, and constraint store S,

- C is globally inconsistent ($T_D \models \forall.(S \rightarrow \neg C)$) only if $C \in \mathcal{C}(D_i)$ is locally inconsistent with the theory T_{D_i} and local store $S_i = S \cap \mathcal{C}(D_i)$ associated with some specialized decision procedure D_i ($T_{D_i} \models \forall.(S_i \rightarrow \neg C)$).
- C is a global consequence of S ($T_D \models \exists.(S \wedge C)$) only if $C \in \mathcal{C}(D_i)$ is a local consequence of the theory T_{D_i} and local store $S_i = S \cap \mathcal{C}(D_i)$ associated with some specialized decision procedure D_i ($T_{D_i} \models \exists.(S_i \wedge C)$)

Whilst correctness of a network of specialized decision procedures is at least theoretically rather unproblematic, completeness is not an easy property to establish. We finesse this problem by simply assuming completeness.

2.5 Answers Returned from Specialized Decision Procedures

The formalization of a decision procedure reflects the underlying theory which defines the constraints. In practice the theory is usually built into the constraint solver as a fixed set of inference rules. Accordingly we view a decision procedure D_i as a theorem prover tuned to proving formulae of a certain class ($\mathcal{C}(D_i)$) from a specific theory T_{D_i}.

An answer returned by a specialized decision procedure must be globally correct. Ideally we would obtain the answer *yes* from a specialized decision procedure if the new constraint was globally consistent, and *no* if it was globally inconsistent. Unfortunately there is, in general, insufficient information available in the local store to support such global decisions, even in case the new constraint is in the class of constraints the specialized decision procedure can decide. Consequently there are three possible answers returned from specialized decision procedure, *yes*, *no* or *unknown*.

Note that we do not require satisfaction completeness for the theories underlying *specialised* decision procedures. The reason is that if local inconsistency cannot be established, the system does not need to conclude that the new constraint is consistent. This can be decided instead by another more general decision procedure.

Because the specialized theory T_{D_i} is a subset of the domain theory T_D, and assuming the local store S_i is a subset of the current constraint store S, local inconsistency always entails global inconsistency. For example suppose an implementation of $CLP(\mathcal{R})$ uses a specialized decision procedure to deal with equations. The local store associated with this procedure also holds only equations, and no inequalities. Suppose the equation $X = 1$ appeared in this local store. The new constraint $X = 2$ can be proved locally inconsistent by the specialized procedure using its local store. No matter what further constraints appear in the current constraint store, the new constraint $X = 2$ is clearly globally inconsistent as well. This is a direct result of our using standard predicate calculus as our formalisation. Since it is monotonic, we can extend the theory and the constraint store freely without invalidating the local proof of inconsistency.

In short, a specialized decision procedure can answer *no* whenever local inconsistency is proved.

However failure to establish local inconsistency does not imply the new constraint is globally consistent. For example a decision procedure for equality might successfully establish the local consistency of a constraint $X = 2$, whilst the global store contains $X > 3$. In this case $X = 2$ is locally consistent but globally inconsistent.

To obtain an answer *yes*, it suffices to prove the new constraint C is already a consequence of the local store. In this case, using the monotonicity of the predicate calculus again, it follows that the new constraint is a consequence of the current constraint store under the domain theory. (If $T_{D_i} \models (S_i \rightarrow C)$ then $T_D \models S \rightarrow C$). Since the operational semantics of $CLP(X)$ entails the consistency of the current constraint store in every computation state, and C is a consequence of the current constraint store, C must be consistent with it.

For example if the local store contains $X = 2$ and $Y = X$ then the new constraint $Y = 2$ is a local consequence, therefore globally entailed, and therefore also globally consistent. Clearly this is a very stringent condition for a *yes* answer which does not arise very often in practical programs. One of the main results of this paper is to significantly loosen the conditions for a *yes* answer from a specialized decision procedure, see section 3 below.

Finally the answer *unknown* is returned from a decision procedure just in case it can answer neither *yes* or *no*.

2.6 Consensus

Specialized decision procedures consist of two components (i) a theory specialized for a class of constraints, and (ii) a local store of constraints. Recall that our aim is to achieve consensus (and thus have local decisions agree with the global decision) but at the same time *avoid* involving costly decision procedures in the network (as far as possible). Therefore the *consensus* has to be inherent in the network. We analyze this interaction by making the relationships (relevant to achieving consensus) between decision procedures explicit. We now consider *what* information

we would like to make explicit.

Consensus requires that two decision procedures, with their own different local stores, can't contradict each other, one answering *yes* whilst the other answers *no*. In this case we have two underlying theories T_1 and T_2 and two local stores S_1 and S_2, and we need to preclude the case that $T_1, S_1 \models \neg C$ and $T_2, S_2 \models \exists C$. Whenever this arises, $T_1 \cup T_2 \cup S_1 \cup S_2$ is inconsistent.

Even if T_1 and T_2 have the same constraint store S, it is possible for S to be consistent with T_1 and T_2 individually but have $T_1 \cup T_2 \cup S$ be inconsistent. The following example illustrates the point. If T_1 is $\{\forall X.f(X) \leq g(X)\}$, and T_2 is $\{\forall X.g(X) \leq f(X)\}$, and S is $\{f(Y) \neq g(Y)\}$, we have a case where $T_1 \cup S$ is consistent, $T_2 \cup S$ is consistent, $T_1 \cup T_2$ is consistent, but $T_1 \cup T_2 \cup S$ is inconsistent.

Consensus is imposed in our framework by the following conditions:

Correctness All the decision procedures associated with a given CLP(D) system, have an underlying theory which is a subset of the domain theory T_D

Completenes New constraints are only accepted by the network of specialized decision procedures if some procedure answers *yes*. By correctness this means that accepted constraints are consistent with the current (global) constraint store. New constraints are only added to local constraint stores if they have been accepted. Consequently the local constraint stores are subsets of the current (global) constraint store, which is consistent ($T_D \models \exists S_D$).

Given these conditions, consensus is a direct consequence of our formalisation in monotonic logic.

3 Optimizing the Design of Composite Decision Procedures

3.1 Loosening the Conditions for a Local *yes* Answer

In this subsection We present three loosened conditions under which a local decision procedure can answer *yes*.

Firstly we can take advantage of existential quantification. Many new constraints include one or more new variables, that do not appear anywhere in the current global constraint store. We shall write $C[\tilde{V}]$ for a constraint involving new variables $\tilde{V} = V_1, \ldots V_n$. A local decision procedure D_i can answer *yes* if, after existentially quantifying over the new variables, it can be proved that the resulting constraint is a consequence of the local constraint store S_i, formally $T_{D_i} \models \forall(S_i \rightarrow \exists\tilde{V}.C[\tilde{V}])$. By monotonicity it follows that $T \models \forall(S \rightarrow \exists\tilde{V}.C[\tilde{V}])$. By the satisfaction completeness of T_D we conclude that $T \models \exists(S \wedge \exists\tilde{V}.C[\tilde{V}])$, and since none of the new variables occur in S we can pull the existential quantification outside the conjunction yielding $T \models \exists.S \wedge C$, thus establishing the global consistency of the new constraint.

As an example we take Gaussian elimination as the specialised decision procedure, whose underlying theory is the theory of equality and whose class of constraints admits linear numeric equations.

Suppose the new constraint is $5 = X + 2 * Y + 3$, where X is a new variable, but Y is not. Then the answer *yes* can be returned since $\exists X.(5 = X + 2 * Y + 3)$ can be proved by making X the subject of the equation $X = 2 - 2 * Y$, and using the axioms: $\forall A, B.\exists X.(X = A - B)$, and $\forall A, Y.\exists Z.(Z = A * Y)$, which states that $-$ and $*$ are functions defined everywhere.

However if neither X nor Y were new variables the unification procedure could not in general answer *yes*, since there could be contradictory constraints in the global constraint store such as $\{Y > X, Y > 0\}$.

Secondly, we introduce a notion of independence. Two constraint stores S_1 and S_2, are independent if the set of variables of S_1 is disjoint from the set of variables of S_2. Under this condition, the local consistency of S_1 and S_2 implies that $S_1 \cup S_2$ is also consistent. Researchers in the area of parallelism also use this property to ensure that when a problem is decomposed that solving the subproblems do not interfere with each other, e.g. restricted AND-parallelism [6]. Independence will be used in section 3.3 below, to keep local constraint stores small.

Independence enables a specialized decision procedure D_i to answer *yes* if the new constraint is provably consistent with its local store S_i even though the proof might "instantiate" variables in S_i. The condition is that S_i is independent of the remaining constraints in $S \setminus S_i$. Formally, the procedure can answer *yes* for a new constraint C if $T_{D_i} \models \exists(S_i \rightarrow C)$. The global consistency of C follows since, by assumption $T_D \models \exists(S_D \setminus S_i)$, and therefore (by monotonicity) $T_D \models \exists S_i \wedge \exists(S_D \setminus S_i)$, and finally, by independence we can pull the existential quantifier outside the conjunction yielding $T_D \models \exists S_D$.

A simple, but important, extension of this result is to allow S_i to share variables with other local stores if they are associated with *more specific* decision procedures in a sense defined in section 3.2 below.

Thirdly, using independence, we can take advantage of satisfaction completeness. If the theory T_{D_i} associated with decision procedure D_i, is satisfaction complete and the constraint store S_i is independent of the remaining constraints $S \setminus S_i$, then D_i can answer *yes* immediately if the new constraint is not locally inconsistent. In this case, therefore, the local decision procedure gives either a *yes* or a *no* answer to every new constraint.

The reason is that if C is not inconsistent with S_i then, by satisfaction completeness, $T_{D_i} \models \exists S_i \wedge C$. Now it follows that $S_i \wedge C$ is globally consistent by the same argument as before.

A first example of this is the use of Herbrand unification for equations which do not involve mathematical functions. For this class of constraints (i.e. equations with uninterpreted functions only), unification is satisfaction complete. Thus any new constraint which is not proved inconsistent by the unification procedure is accepted as globally consistent. In $CLP(R)$ accordingly, equations involving only "non-solver" variables, are handled by the unification in the logic programming engine and are never passed to the specialised arithmetic solvers.

As another example, the theory underlying Gaussian elimination is satisfaction complete for the class of mathematical equations. As long as the set of variables in the local constraint store associated with Gaussian elimination is disjoint from the variables in the current Simplex tableau, for a new equation which also shares no variables with the Simplex tableau, independence can thus be used to enable Gaussian elimination to decide new constraints without resorting to the Simplex algorithm.

Returning to an earlier example $5 = X + 2 * Y + 3$, even if X is not a new variable, as long as the local store shares no variables with the Simplex tableau, it can be treated using Gaussian elimination alone. If the equation is not inconsistent with the local store associated with Gaussian elimination, the procedure simplifies it to $X = 2 - 2 * Y$ (see section 3.4 below) but but, since the local store is independent, and the theory underlying Gaussian elimination is satisfaction complete for equations, instead of *unknown* it returns the answer *yes*.

3.2 Using Specialized Procedures to Decide Constraints

First we present an abstract algorithm for determining satisfiability given a network of cooperating decision procedures. Its formulation here as a distributed algorithm is merely done to simplify the discussion. Short cuts are achieved by broadcasting the result returned by one decision procedure and thereby interrupting other decision procedures and saving them any further work.

The behaviour of each decision procedure can be described in terms of a single "check" operation which invokes the decision procedure on the new constraint with the local store. Four things can happen:

- The procedure returns the answer *yes*. In this case the check succeeds and broadcasts its result to all the other checking processes.
- The procedure returns the answer *no*. In this case the check fails and broadcasts its result to all the other checking processes.
- The answer is *unknown*. In this case the check suspends.
- The checking process is interrupted by a broadcast. In this case it terminates with the same result as that broadcasted.
- A suspended process that recieves a broadcast decision terminates with that decision.

The final result is either that all the checks succeed, or they all fail. Since we assume the network to be complete with respect to the (satisfaction complete) domain theory (see section 2.4 above), they cannot all suspend. The consistent behaviour of all the check operations is guaranteed by the consensus of the decision procedures and their local stores.

If the checks fail is the new constraint globally inconsistent, and it is not admitted. Operationally, such a result causes the system to start backtracking. If the checks succeed, then the new constraint is globally consistent. It is then admitted and added to certain local stores, as described below.

Of course one does not wish to run all the decision procedures in parallel. Instead what is desired is that calls to the expensive decision be avoided. To this end we use a taxonomy based on subsumption. A decision procedure D_1 is said to be *more specific* than a decision procedure D_2 if

1. the theory T_{D_1} of D_1 is a subset of the theory T_{D_2} of D_2,

2. The class $\mathcal{C}(D_1)$ is a subset of $\mathcal{C}(D_2)$

3. $S_1 \subseteq S_2$. In this case we say that the constraint stores are *admissible*.

For a new constraint apply the decision procedures earliest in the ordering first, and only if the answer is *unknown* move up the ordering. This sequential use of decision procedures also supports constraint simplification (see section 3.4 below).

3.3 Distributing Constraints to Local Stores

If every accepted constraint C is added to every local store for which C is in its class of constraints, then by assumption the completeness of the network of decision procedures is preserved. However it is possible to retain completeness without adding every constraint to every appropriate store.

Firstly let us consider new constraints that share no variables with the global constraint store. If S and C share no variables, then by indepedndence as we argued above, they are globally consistent $T_D \models \exists S \wedge C$, if and only if they are independently consistent $T_D \models \exists S$, and $T_D \models \exists C$.

Thus there is no need to use any (local) constraint store in establishing the consistency of C. Similarly if a new constraint C_2 is added, which shares no variables with C, then the presence of C in any (local) constraint store during the checking of C_2 is unnecessary. Consequently, until two constraints are added which share variables, there is no need to add any constraints to any constraint store.

This independence result can be used to partition the local stores associated with each specialized decision procedure into disjoint local stores which share no variables. When a new constraint is checked which only shares variables with one local store in the partition, then it need only be checked against that local store and no other.

Furthermore the partitioning significantly increases the potential for global independence of a local store. For example although the local constraint stores associated with Gaussian elimination may indeed share variables with the Simplex tableau, individual components of the partition may be quite independent of it. In this case consistency with a single component suffices to establish the global consistency of any new constraint whose variables are shared only with that component.

Even more interestingly, we can use satisfaction completeness to extend the partitioning beyond a single specialized decision procedure. Consider a partitioning of the global constraint store S_D. Suppose all the constraints in some component S_k belong to the class $\mathcal{C}(D_i)$, and that D_i has a satisfaction complete underlying theory T_{D_i}, for the class $\mathcal{C}(D_i)$ of constraints. Then only the decision procedure D_i needed be used for checking a new constraint $C \in \mathcal{C}(D_i)$ which only shares variables with S_k. Either it is locally consistent, which by independence and satisfaction completeness entails global consistency, or it is locally inconsistent. In each case the local decision is sufficient to produce a *yes* or *no* answer. If a new constraint shares no variables with S_{i_k}, then it can be checked by all other specialized decision procedures against their local stores, independently of the constraints in S_{i_k}.

The advantage of partioning for increasing the number of local decisions has been established. To ensure the local stores are partitioned it is necessary when adding a new accepted constraint to add it to the component with which it shares variables and no other component of the partition. As long as the new constraints fall into the current components, the arguments above based on indepence and satisfaction completeness prove that the behaviour of the network of decision procedures remains correct and complete. Therefore in this case there is no need to add the new constraint to every local store which admits constraints of that class.

This result is used in $CLP(R)$ to add new Herbrand equations only to the unification environment of the logic programming engine and not to any of the solvers, in case there are no "solver" variables in the equation (see above, section 3). Of course it is vital for preserving the independence of the unification environment from the other local constraint stores.

However sooner or later a constraint will be added that does not fall into one of the current components. Either it will share a variable with more than one component, or else it may belong to a single component but fail to belong to the appropriate restricted class of constraints.

In the first case the check is performed against a local store comprising the union of the affected partitions. Subsequently this union, together with the new constraint, forms a single partition.

Sometimes the new constraint shares variables with a previously independent component and with the local store associated with other decision procedures. This also happens if the new constraint "belongs" to a single component, but not to its restricted class of constraints. In both cases the new constraint must be checked by further decision procedures. However each decision procedure must check the new constraint against the union of its local store and the (previously independent) component. Subsequently the whole partition must be added, with the new constraint, as if all the constraints were new. In $CLP(R)$ this occurs, for example, when a new equation causes "solver" and "non-solver" variables to be unified.

We conclude this subsection by describing the distribution of constraints between local stores according to the subsumption taxonomy of decision procedures.

The completeness of the network of specialized decision procedures is not jeopardized if a new constraint is omitted from the store associated with a more specific decision procedure. The reason is that although the more specific procedure will reach fewer decisions, in case no decision is reached a more general procedure will be used, whose constraint store is complete. This result is used in the implementation of $CLP(R)$ to ensure a new constraint is only (ever) added to one local constraint store. However the duplication of constraints between more specific and more general decision procedures has benefits for future constraint checking. The more constraints held in the local store associated with a more specific procedure, the more decisions it can reach. An example illustrating the advantage of duplicating constraints is given in section 4.2 below.

3.4 Simplifying Constraints

The decision procedures could be implemented using simplification [12]. Instead of just trying to prove $T, S \models \neg C$ or $T, S \models \exists \tilde{V}.C[\tilde{V}]$, the procedure derives a simpler formulae $CSimp[\tilde{V}]$ equivalent to $C[\tilde{V}]$ in the sense that $T, S \models C[\tilde{V}] \equiv CSimp[\tilde{V}]$.

If the constraint is inconsistent, $T, S \models \neg C[\tilde{V}]$, then this is made explicit during simplification: $CSimp[\tilde{V}]$ is just $false$. In this case the procedure returns the answer *no*. If the constraint is a logical consequence, $T, S \models \exists \tilde{V}.C[\tilde{V}]$, then the procedure must explicitly check whether $\exists \tilde{V}.CSimp[\tilde{V}]$ is a consequence of the underlying theory T. In this case it answers *yes*. If $\exists \tilde{V}.CSimp[\tilde{V}]$ is neither *false*, nor a consequence of T, then the procedure answers *unknown*.

Decision procedures which perform simplification can produce useful information even though their answer is *unknown*. Most commonly this is when a specialized decision procedure yields a specific value for a variable. When a constraint checked by a specialized decision procedure becomes simplified, then if the answer is *unknown*, instead of checking the original constraint with the other procedures, the simplified constraint is now checked.

Suppose for example the global constraint store included $X = Y + 1$ and $X > 2$. The inconsistency of the new constraint $2 * X = Y$, could be detected by the Simplex procedure. Alternatively however Gaussian elimination can simplify the new constraint to $X = -1$ and $Y = -2$, and the inconsistency can now be detected by the built-in inequality predicate of the underlying logic programming system.

The use of specialized decision procedures for constraint simplification can be used to reduce the number of distinct variables in local stores associated with the more general decision procedures. This simplifies constraint solving and increases independence. The technique requires certain equations of the form $Var = Term$ to be omitted from the local constraint stores associated with more general procedures, such as the Simplex. Completeness is retained by always simplifying constraints involving Var, to constraints involving $Term$. Though intuitive, this technique depends upon simplification being always carried out before constraint solving, and therefore somewhat restricts the framework presented in this paper. In most CLP systems the unification decision procedure is used for simplification of constraints in the way we just described. Gaussian elimination could also be utilised in the same way.

3.5 Detection of Redundant Constraints

The decision procedures naturally support a facility to recognise and prevent the addition of redundant constraints. If a new constraint C has no new variables, and some decision procedure returns the answer *yes*, then C is redundant. Since it is globally consistent (i.e. $T_D \models \exists S_D \wedge C$), and C has no variables, therefore it is independently consistent (i.e. $T_D \models C$).

On the other hand, our framework makes full use of the capacity of any specialized decision procedure D_i to detect redundancy. If $T_{D_i} \models \forall.(S_i \rightarrow C)$ for some new (redundant) constraint C, then the decision procedure immediately answers *yes*.

3.6 An Example

We now give an example in the domain of real linear arithmetic. Consider a two-vertex network consisting of a decision procedure for *equalities*, call it CEq and a decision procedure for *equalities and inequalities*, call it CInEq. The theories underlying CEq and CInEq are T_{CEq} and T_{CInEq} respectively. Thus we have that $T_{CEq} \subseteq T_{CInEq}$. Suppose the global constraint store is $\{X =$

$4, Y = 2, X > 3, Z > 4\}$. The local store for CEq is $\{X = 4, Y = 2\}$. The constraint store for CInEq contains all the constraints: $\{Y = 2, X = 4, X > 3, Z > 4\}$. We now present three cases. These are the use of local consistency (resp. inconsistency) in quickly determining *global* consistency (resp. inconsistency) and the case when the specialized decision procedure returns unknown.

1. Let the constraint to be incrementally conjoined be $X = Y$ then CEq detects $X = 4 \wedge Y = 2 \rightarrow \neg(X = Y)$ and thus global inconsistency.

2. Let the constraint to be incrementally conjoined be $X = W$. Then CEq detects that $X = 4 \wedge X = W \rightarrow W = 4$. Notice that W is a new variable. Since $\exists W.(W = 4)$ is a simple consequence of the equality theory, CEq answers *yes*, proving global consistency.

3. Let the constraint to be incrementally conjoined be $Z = 2$. CEq can infer neither $Z = 2$ nor $\neg(Z = 2)$ so it waits (as the answer locally is *unknown*). But CInEq infers $\neg(Z = 2)$ and global inconsistency is detected.

4 Relationship to Existing Work

4.1 The CLP($\mathcal{R}$) System

CLP($\mathcal{R}$) is an instance of the CLP scheme in the domain of uninterpreted functors over real arithmetic terms. Its decision procedure for arithmetic constraints is implemented as a multi-layered combination of

1. the Herbrand unifier,

2. a Gaussian elimination procedure,

3. a Simplex algorithm,

The handling of arithmetic constraints is summerized as follows. Linear inequalities are decided using a modified Simplex algorithm. Linear equalities are sent to a Gaussian elimination procedure. However, if a linear equality affects any linear inequalities it is immediately sent to the Simplex solver. The Herbrand unifier deals with simple arithmetic constraints of the following form.

- an equation between two non-arithmetic variables,
- an equation between a non-arithmetic variable and an arithmetic variable,
- an equation involving an uninterpreted functor,
- an equation or inequality between two numbers, and
- an equation between a non-arithmetic variable and a number [15].

4.2 The Cooperating Decision Procedures of CLP($\mathcal{R}$)

We now analyze the decision procedures of CLP($\mathcal{R}$) to see how they achieve consensus.

In terms of our framework we view the Herbrand decision procedure has having two different local stores, whose constraints involve disjoint sets of variables. This idea was discussed in section 2.5 above. Equalities involving arithmetic variables, which also appear in local stores associated with the Gaussian or Simplex procedures, are kept in one store. Equalities involving non-arithmetic variables are held in the other.

For non-arithmetic variables the equality theory is satisfaction complete, so if unification fails the inconsistency of the constraint can be deduced (see section 2.5). Unification is a simplication procedure, and in CLP($\mathcal{R}$) it is the simplified constraint that is passed on to the other solvers, see section 3.4 above.

If constraints cannot be solved by the Herbrand unifier, they are passed to the less efficient numerical decision procedures. Numeric equations are next checked by the Gaussian elimination procedure. The theory underlying Gaussian elimination is satisfaction complete for equalities. For equations sharing no variables with other constraints held in the Simplex store, this enables Gaussian elimination to return success if it doesn't explicitly fail. However for equations involving variables appearing in the Simplex store local consistency no longer implies global consistency. Therefore in CLP($\mathcal{R}$) very sophisticated analysis of the variables is carried out, including simplification using equations in the Gaussian store, to decide if the new equation should be passed up to the Simplex solver or not. This analysis reduces, in our framework, to an attempt to get the answer *no* or *yes* from the Gaussian decision procedure.

Finally, if neither unification nor Gaussian elimination can be used to decide a new equality constraint, it is passed to the Simplex solver. Inequalities, which fall outside the syntactic class of constraints handled by unification and Gaussian elimination are also passed direct to the Simplex solver. Again the underlying theory for the Simplex solver is satisfaction complete, so if inconsistency cannot be proved, the system has soundly established global consistency.

The operational behaviour of CLP($\mathcal{R}$) fits into our framework, and the rationale for the global decision procedure corresponds closely to that described in section 2.1 above. The completeness and soundness of CLP($\mathcal{R}$) is thus justified in our framework.

4.3 Some Suggested Modifications to CLP($\mathcal{R}$)

There are more shortcuts which CLP($\mathcal{R}$) fails to exploit. In CLP($\mathcal{R}$) there is no attempt to use unification for solving equations involving numeric terms. However this is a possible check since if the terms are identical, up to the occurrence of new variables, then the Herbrand unifier could immediately answer *yes*.

In the framework of this paper it is clear that the more constraints locally available to a weak, but efficient, decision procedure, the more chance it has of producing a *yes* or *no* answer and short cutting further checking. Thus we would expect the efficiency of CLP($\mathcal{R}$) to be enhanced by copying equalities from the Gaussian store to the unification store. For example equalities like $Y = X + 1$ could be used by the unification procedure to simplify incoming

constraints. If all occurrences of Y could thereby be eliminated from the Gaussian store, the total number of arithmetic variables could be reduced.

The elimination of redundant constraints is an important issue for efficiency. The recognition that new constraints are redundant, so that they are never added to any constraint store, is achieved naturally within our framework (see section 3.5 above). Although some techniques using independence and subsumption criteria to identify and propagate useful information are employed in CLP($\mathcal{R}$) they do not seem to have been systematically analyzed and used.

4.4 Other Methods

The technique of using specialization has also been tackled by Bruynooghe et al. [2]. Their approach relies on analyzing instantiation patterns and using a symbolic trace to derive control information. Our method is more general in that we can *incorporate* specialized decision procedures whereas their method concentrates essentially on taking advantage of evaluation.

5 Conclusion

Since general constraint solvers are usually quite expensive and real world problems quite often involve just certain classes of constraints [10, 11, 13] one way to increase efficiency is to use specialised decision procedures for handling specialised classes of constraints.

In this paper we have explored the problem of how to use the network as efficiently as possible. A given problem should be decided by the simplest procedures which can do the job. Moreover the number of constraints checked for consistency with each new constraint should be kept to a minimum.

By formalising decision procedures as theorem provers, we have set up a framework in which the problem and requirements can be clearly specified. A number of domain-independent properties of decision procedures, relevant to efficient constraint solving have been identified. A fundamental property is *consensus*, which must hold between all decision procedures and local constraint stores in a network. The class of constraints for which a decision procedure is *satisfaction complete* [14] is also important. These properties enable certain short cuts to be taken which save using unnecessary decision procedures.

The architecture of the CLP($\mathcal{R}$) decision procedure has been shown to fit the framework we introduce and make use of many of the optimisations though, interestingly, not all.

The ideas presented here are being used in the development of a new CLP platform at ECRC.

Acknowledgements

The ideas in this paper were discussed with Alex Herold, Thom Frühwirth, Eric Monfroy and Volker Kuechenhoff. We thank them for their many valuable comments.

References

[1] R. Brachman, V. Gilbert and H. Levesque, "An Essential Hybrid Reasoning System: Knowledge and Symbol Level Accounts of Krypton", Proceedings of the 9th IJCAI, 1985.

[2] M. Bruynooghe et al., "Improving the Efficiency of Constraint Logic Programming Languages by Deriving Specialized Versions", International Workshop on Processing Declarative Knowledge, Kaiserslauten, Germany, July 1–3, 1991. Published as Springer-Verlag LNCS 567.

[3] K.Clark, "Negation as Failure", in "Logic and Databases", ed. H. Gallaire and J. Minker, Plenum Press, 1977.

[4] G. Collins, "Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition", LNCS, No. 33, Springer-Verlag, New York, (1975).

[5] J. Davenport and J. Heintz, "Real Quantifier Elimination is Doubly Exponential", Journal of Symbolic Computation, (1988)5, 29–35.

[6] D. DeGroot, "Restricted AND-Parallelism", Proceedings of the International Conference on Fifth Generation Computer Systems 1984, North Holland, 1984, pp. 471–478.

[7] F. Donini, M. Lenzerini, D. Nardi and W. Nutt, "Tractable Concept Languages", Proceedings of the 13th IJCAI, Sydney, Australia, August, 1991.

[8] Y. Deville and P. Van Hentenryck, "An Efficient Arc Consistency Algorithm for a Class of CSP Problems", Proceedings of the 13th IJCAI, Sydney, Australia, August, 1991.

[9] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf and F. Berthier, "The Constraint Logic Programming Language CHIP", Proceedings of the 2nd International Conference on Fifth Generation Systems, Tokyo, November 1988, pp. 249–264.

[10] M. Dincbas, H. Simonis and P. Van Hentenryck, "Solving the Car-Sequencing Problem in Constraint Logic Programming", Proceedings of the 1988 European Conference on Artificial Intelligence, Munich, West Germany, August 1988.

[11] M. Dincbas, H. Simonis and P. Van Hentenryck, "Solving a Cutting-Stock Problem in Constraint Logic Programming", Fifth International Conference on Logic Programming, Seattle, August 1988.

[12] T. Frühwirth, "Constraint Simplification Rules", ECRC, Draft, March, 1992.

[13] N. Heintze, S. Michaylov and P. Stuckey, "CLP($\mathcal{R}$) and Some Electrical Engineering Problems", Proceedings of the 4th International Conference on Logic Programming, Melbourne, 1987, pp. 675–703.

[14] J. Jaffar and J-L. Lassez, "Constraint Logic Programming", Proceedings of the 1987 ACM Symposium on Principles of Programming Languages, Munich, January 1987, pp. 111–119.

[15] J. Jaffar, S. Michaylov, P. Stuckey and R. Yap, "The CLP($\mathcal{R}$) Language and System", IBM Research Report RC 16292, November, 1990.

[16] L. G. Khachiyan, "A Polynomial Algorithm in Linear Programming", Soviet Mathematics Doklady, Vol. 20, 1979, pp. 191–194.

[17] H. Kirchner and C. Ringeissen, "Combining Unification Problems with Constraint Solving in Finite Algebras", Presentation at WCLP'92, Marseille-Luminy, February 12–14, 1992.

[18] G. Nelson and D. Oppen, "Simplification by Cooperating Decision Procedures", ACM TOPLAS, Vol 1, No. 2, October 1979, pp. 245–257.

[19] G. Smolka, "Records for Logic Programming – Constraints on Feature Trees", Presentation at WCLP'92, Marseille-Luminy, February 12–14, 1992.